Channels: Runtime System Infrastructure for Security-typed Languages

Boniface Hicks*
Saint Vincent College
fatherboniface@acm.org

Timothy Misiak and Patrick McDaniel
Penn State SIIS Lab
tgm117@psu.edu, mcdaniel@cse.psu.edu

Abstract

Security-typed languages (STLs) are powerful tools for provably implementing policy in applications. The programmer maps policy onto programs by annotating types with information flow labels, and the STL compiler guarantees that data always obeys its label as it flows within an application. As data flows into or out of an application, however, a runtime system is needed to mediate between the information flow world within the application and the non-information flow world of the operating system. In the few existing STL applications, this problem has been handled in ad hoc ways that hindered software engineering and security analysis. In this paper, we present a principled approach to STL runtime system development along with policy infrastructure and class abstractions for the STL, Jif, that implement these principles. We demonstrate the effectiveness of our approach by using our infrastructure to develop a firewall application, FLOWWALL*, that provably enforces its policy.*

1 Introduction

Security-typed languages (STLs) provide a means of *verifiably* ensuring that an application will enforce its security policy. This is accomplished analogous to mandatory access controls (MAC), but *within* applications. In a MAC operating system, all processes and system objects are labeled and a reference monitor ensures all security sensitive operations obey the labels relative to a system policy. In STL applications, all data is labeled and the type checker ensures that all data operations obey the labels, relative to an application policy. STLs can accomplish this fine-grained security checking efficiently, because much of the checking can be completed at compile-time. For the remaining checks, the STL compiler ensures the required dynamic checks are in place in the code. In this way, the STL compiler enforces information flow properties (data confidentiality and integrity) from end to end (data input to output) within an application.

A critical element needed for implementing security policy using STLs has been neglected, however. Thus far, the primary focus has been to implement security policy by 1) placing labels statically on code and 2) by defining a label semantics (an application policy) using a principal hierarchy. To implement security policy in real applications, however, policy must also be implemented at a third policy enforcement point: in a *runtime system* (RTS). The runtime system is responsible for mediating all communications between the information-flow world within the application and the (possibly) non-information flow world outside the application. *Without runtime infrastructure to handle dynamic labeling of I/O, STLs can only be effective for analyzing closed systems* (with no inputs or outputs).

Real programs are seldomly closed systems, however. They must interact with a world outside themselves, receiving data from and sending data out on various channels. Consider an email client that must interact with a remote mail server as well as a local user, various files and databases. Another example is a network firewall that must interact with the networking subsystem and may also save audit logs to files.

STLs can perform a critical service in provably ensuring *noninterference*, i.e. that secret flows and public flows will remain separate everywhere *within* an application. They cannot, however, automatically reason about the security of data as it crosses the application boundary to and from the system. A runtime system is needed to handle these security decisions, governing whether the data arriving on a socket should be labeled secret or public or `alice`-data or `bob`-data (in fact, labels may even be much more expressive than this) or determining whether secret data may safely be output to a particular file. In fact, these are critical decisions; the rest of the label checking is moot if this step is not handled correctly.

*This material is based upon work supported by Motorola through SERC and the National Science Foundation under Grant No. CCF-0524132, "Flexible, Decentralized Information-flow Control for Dynamic Environments" and CNS-0627551, "CT-IS: Shamon: Systems Approaches for Constructing Distributed Trust". Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, Motorola or SERC.

This has been recognized, de facto, in the construction of existing STL applications [1, 5, 8, 13, 4]—each implemented its own runtime infrastructure to cover its own particular policy for labeling I/O. It is notable that these runtime infrastructures could not be reused from one application to the next. This is because the labeling of I/O can be quite subtle, dependent on several factors, such as what system resources are used by the application, what system security mechanisms are available for these resources and what authentication protocols are used to identify the resources and the data they carry. In other words, runtime system infrastructure must be *specialized* for different environments and applications.

In this paper, we give principles to guide the development of application-specific runtime system infrastructure for STL applications. Our principles provide for modularized, specialized runtime systems that can be configured and controlled through high-level policy. As a manifestation of these principles, we provide a `Channel` abstraction for the most mature STL, Jif, a variant of Java. The `Channel` facilitates the mapping of policy onto I/O channels and thus serves as the basic building block of a runtime system. It can be implemented and extended in different settings according to the individualized needs of different applications. We also provide a high-level policy infrastructure for activating and configuring `Channel`s.

We evaluate our approach by using our RTS tools to build FLOWWALL, a basic, packet-filtering network firewall [3]. As part of FLOWWALL, we construct a new `Channel`, a `PacketChannel`, which handles the inputting and outputting of all network packets. The `PacketChannel` is constructed in such a way that in order for a packet to pass through the FLOWWALL, it must flow from its source to its destination address. The Jif compiler provably ensures that all flows obey the label semantics defined in an application flow policy. By automatically compiling this application flow policy such that it is isomorphic to the firewall policy, we are able to implement a firewall that verifiably enforces its firewall policy.

The result of our investigation shows that the `Channel` is effective for implementing high-level security policy in the runtime infrastructure. The `Channel` integrates well with STLs because it can be incorporated into an STL policy infrastructure such that it can be configured and controlled at a high level. It also facilitates reusability, because it utilizes a pluggable interface for various protocol-based, authentication-based and other data-specific labeling mechanisms. Finally, it aids STL software engineering, because it provides an intuitive interface for handling a problem that repeatedly faces STL programmers.

In Section 2 we show how STLs facilitate the implementation of information flow security in applications, while also describing the challenges of handling dynamic labeling of I/O. This leads us to some principles for developing runtime infrastructure. We then describe how these principles can be manifested through our `Channel` abstraction, presented in Section 3. In Section 4 we show how the `Channel` can be used to implement infrastructure for FLOWWALL. We give an evaluation FLOWWALL focusing on the effectiveness of our principles in Section 5 . In Section 6 we consider related work. We conclude in Section 7.

Listing 1. Keeping statistics on incoming packets.

```
1 public class Stats[label L] {
2   int{L} sshCount;
3   public Stats() { sshCount = 0; }
4   public void checkPacket{L}(Packet pkt)
5         where {pkt} <= L
6   { if (pkt != null && pkt.destPort == 22)
7       sshCount++;
8 }}
```

2 Security for STL Applications

Using an STL, a programmer is able to provably implement high-level security policy in applications. Specifically, establishing overall security properties in security-typed applications depends on the security policies established in three separate parts of the application: 1) the dynamic labeling of inputs and outputs, 2) the static labeling of code in the application, and 3) the relationships between labels (label semantics). While it was well understood that (2) and (3) are key parts to implementing system policy in security-typed languages, the importance of (1) has not previously been studied; only ad hoc infrastructure has been built for prior applications.

The power of the STL compiler consists primarily in its ability to automatically limit the labels that may be placed statically on code (2) such that they must honor the labels in (1) with respect to how they may licitly flow, as defined by (3). In other words, so long as the security analyst can verify that inputs and outputs are properly labeled (1) and the label relationships are correctly established (3), the STL compiler will automatically check the rest. The problem is that without careful design of the runtime system and without an accompanying high-level policy, these three dimensions of the security policy may be scattered throughout the code.

2.1 Automatic analysis performed by STL compiler

One of the compelling features of STLs is their modularity [11]—they can ensure security through composition of secure modules. Modules can be separately type-checked for security and compiled, then later combined to make secure applications. The more generally the module's security properties can be expressed, the more widely it can be used

in different applications.

Developing a basic module Consider the Jif code in Listing 1. This small module can be used to keep statistics on network packets as they pass through an application such as a network firewall. It is parameterized by a label `L` with the annotation [**label** `L`] on line 1 and has a single member variable, `sshCount` on line 2, which is guaranteed to be protected at the level of `L` (indicated by the annotation {`L`} on `sshCount`'s type). In this context, "protected" refers to information flow properties such as confidentiality and integrity. This variable keeps a running count of the packets that are being sent to port 22.

Parameterizing a class with a label allows (but does not require) that label to be used within the class. The label parameter must be instantiated when an object of the class is created. For example, the programmer may instantiate a `Stats` object that is visible only to the firewall administrator as follows:

```
Stats[{admin:}] statsObj = new Stats[{admin:}]();
```

A key advantage offered by parameterized classes is the separation of policy from the class; the class makes no restriction on `L`. The only restriction comes from calling the method `checkPacket`. When called, information about the parameter `pkt` will implicitly flow into `sshCount`, therefore Jif requires the programmer to place a restriction on `checkPacket` to ensure `sshCount` will protect the information in `pkt`. This restriction is expressed on the method header with the constraint, **where** {`pkt`} <= `L` (line 5). At the call sites for `checkPacket`, where `L` and {`pkt`} must already be instantiated, the Jif compiler ensures this constraint holds for those particular label instantiations. Whether the constraint holds is determined by the label semantics encoded in the principal hierarchy, defined in a high-level policy external to the application.

Note the advantages of compositionality here. `Stats` can be designed apart from any particular application, and when it is inserted into an application, the STL compiler ensures it will not weaken the security properties of the application. It may restrict the flows in the application, but it cannot introduce any leaks.

Adding system I/O On the other hand, when system inputs and outputs are added to code, they can modify the security policy of the application. Let us consider the challenges this introduces. We add a print method to `Stats` to output the current count, shown in Listing 2. Should this be legal? No. The programmer explicitly ensured that the `sshCount` label (Listing 1, line 2) was secret enough to protect information about packets. The method `printStats` cannot simply print the value to standard output, without discerning whether standard output is secret enough to protect the value.

Listing 2. A faulty first attempt.

```
1 public void printStats()
2 { System.out.println("SSH count: " + this.sshCount); }
```

Listing 3. A valid approach.

```
1 public void printStats{L}() {
2   final label userL = Runtime.userLabel();
3   PrintStream[userL] out = Runtime.stdout(userL);
4   if (L <= userL)
5     if (out != null) out.println("SSH count: " + this.sshCount);
6 }
```

Labeling I/O with the process owner A straight-forward fix is to retrieve standard output from a runtime system that implements a particular policy. For example, a reasonable policy defines the standard output to be as secret as the UNIX user who ran the program. This gives rise to the code in Listing 3. This is not a bad solution. The `Runtime` keeps track of the process owner (initialized at program start up) and stores a corresponding user label. When retrieving the standard output stream (line 3), it requires the stream to be parameterized by the user label, `userL` (indicated with the code `PrintStream[userL]`). Whenever something is printed on the stream, a dynamic check is made, querying the label semantics to be sure that `userL` is sufficiently secret to protect the data in `sshCount`, i.e. that `L` <= `userL` (line 4). This approach is essentially the approach offered by Jif's default runtime system.

Notice how policy is encoded in the construction of the `Runtime` class in Listing 3. Making standard I/O as secret as the user running the application is an approximation of an information flow policy. However, this approximation may not hold in different settings: the terminal window may be in plain sight, in which case it should be considered public. In other settings the secrecy of the terminal window may be determined from the windowing system. *This facet of runtime infrastructure (indeed all runtime infrastructure) implements application security policy and should be configured based on the security goals and assumptions of the particular application.*

The process owner approach is also limited in another way. It prevents the possibility that the output stream could be authenticated to a higher secrecy level. For example, although the application is run by `alice`, she might have special privileges allowing her to see information on packets. To use those privileges, however, she must dynamically *authenticate* herself by providing a password. This requires the addition of an input stream. Furthermore, the data retrieved through the input stream must be able to change the security label on the output stream to reflect a valid authentication. This gives rise to the final version of `printStats`

Listing 4. Robust approach using Channels.

```
1 public void printStats{L}() {
2   Channel stdio = Policy.getChannel("stdio",null);
3   Policy.authenticate(stdio,"stats");
4   final label outL = stdio.getNextOutputLabel();
5   if (L <= outL)
6     if (stdio != null)
7       stdio.put("SSH count: " + this.sshCount, outL);
8 }}}
```

using more advanced data structures to handle labeled I/O channels.

A new construction Listing 4 gives our solution to this challenge. In this final version, policy decisions are deferred to a `Policy` module. The `Policy` provides methods for retrieving system channels, each in its own `Channel` object (line 2) and authenticating them (line 3). The `Channel` abstraction contains a pair of input and output streams. The input stream delivers `LabeledObjects` which package together the next object on the input stream with its label. The output stream can be queried to determine what level of protection it can ensure for the next object. This label can be used to determine whether it is possible to output a given object on the stream. The `Channel.put` method requires that the label on its first parameter (the object that will be output) be dominated by the second parameter (the label it expects for its next output, which may always be retrieved by a call to `Channel.getNextOutputLabel()` as on line 4).

As a result, the Jif compiler requires the programmer to guard the call to `put` with the dynamic check **if** (`L <= outL`) (line 5). We present the `Channel`, `Policy` and `LabeledObject` classes in more detail in Section 3.

The key advantage to our solution is that it adds a layer of indirection, isolating policy decisions to a policy module that can be configured external to the program. In a spirit similar to the policy handling in the `checkPacket` method, the labels and channels need not be determined here, but only checked at runtime for certain relationships. The Jif compiler ensures all needed runtime checks are in place. Policy decisions such as whether to allow standard I/O to be used at all, how it will be labeled, and what authentication to allow for this channel, are lifted out to a separate module that can be controlled and configured along with the rest of the policy for the application.

2.2 Runtime system principles

To accurately determine the security properties of the entire system in which this code is executed, it is necessary to analyze the labels dynamically placed on system objects (the packets and output stream in this case) and the relationships between labels (which will determine the result of the dynamic policy check **if** (`L <= Lout`)). If the dynamic labeling of system objects or relationships between labels do not properly reflect the system security context, the system will fail to meet security goals, despite its automatically checked objects.

These critical security decisions should not be hidden in application code, but isolated into separate modules and configured as part of an external, high-level policy. To this end, we have found the following principles to be effective for moving security policy decisions regarding the dynamic labeling of I/O out of main application code and into runtime modules governed by high-level policy. These principles serve as the requirements for our runtime system and its use. We provide further rationale for the principles as we explain the additional infrastructure we provide (Section 3) and we return to these principles after presenting our FLOWWALL application (Section 4) to show how they were effective. For now, we simply list them.

Principle 1 *Isolate dynamic labeling* by placing the code for dynamic labeling of system objects (inputs and output channels) into the runtime system infrastructure.

Principle 2 *Limit the runtime API* such that it is carefully controlled and as minimal as needed for the applications.

Principle 3 *Customize the semantic granularity* of dynamic system labelings by ensuring that the security context determined for inputs and outputs corresponds with the desired granularity of control in the application.

Principle 4 *Configure runtime labeling through high-level policy* by governing what `Channels` and authentication may be used based on high-level policy.

We define the term *semantic granularity* in Principle 3 to refer to the amount of semantic structure an object has. For example, a stream of bytes has less semantic granularity than when those bytes have been assembled into IP packets or emails. This is not a strict measurement but intended to reflect the insight that the security properties of inputs and outputs can often depend on the semantics of the data, and a datum's semantics cannot be understood until the data is parsed to a higher semantic granularity.

3 Runtime system

In this section, we present a new runtime system (RTS) infrastructure, designed and implemented according to the principles presented in Section 2.2. Figure 1 provides an overview of our infrastructure for compiling policy into a Jif application. In this system, the programmer is responsible for developing Jif application code. If the application requires specialized runtime components (our FLOWWALL requires a component to interact with the network packet

Listing 5. Jif signature API for Channels.

```
public abstract class Channel[label L] {
  public abstract LabeledObject[L] getNextObject{L}();
  public abstract label{L} getNextOutputLabel{L}();
  public abstract void put{*lbl}(Object{*lbl} obj, label{*lbl} lbl);
}
```

stream using a special `libipq` library, e.g.) for communicating with the host system, these must also be provided along with `Channel` interfaces for using them. The Jif flow policy and RTS policy can be customized by the application deployer. The RTS policy determines what channels can be activated as well as (optionally) configuring how they do labeling and authentication.

Our two key contributions are the `Channel` abstraction and the RTS policy compiler; they work in concert with the existing Jif compiler [10] and Jif policy compiler [6, 5]. The RTS compiler produces a `Policy` object based on the policy it has been given. The `Channel` abstraction can be extended to implement different kinds of channels. The `Policy` class controls what `Channels` can be used when executing the application and may configure some labeling and authentication schemas used by the `Channels`. The `Policy` class should only be generated automatically from a high-level policy and then linked into a final application when it is executed. In this way, the channels and authentication used by an application can be controlled through high-level policy while still providing for separate compilation of application modules.

3.1 Channels

The basic `Channel` API is shown in Listing 5. `Channels` cannot be created directly (the API disallows this); they can only be instantiated through the `Policy` class. The `Policy` class is configured using high-level policy and may include or exclude the methods to instantiate various `Channels`. This serves to separate policy specification from its implementation in the application.

A `Channel` delivers labeled objects from the system to the application (inputs) and from the application to the system (outputs). `Channel.getNextObject` returns an object packaged with its label in a `LabeledObject`. For outputs, `Channel.getNextOutputLabel` returns a label and `Channel.put` only accepts outputs with lower security requirements than expressed by that label.

Past experience building STL applications exposes the main challenges for developing a channel abstraction that is both sufficiently expressive and sufficiently general to be useful in a wide variety of settings. These challenges include that 1) labeling of I/O depends on the security mechanisms offered by specific environments (contrast SELinux mandatory access controls with UNIX ACLs on files with authenticated sockets, etc.). 2) Labeling of data sometimes

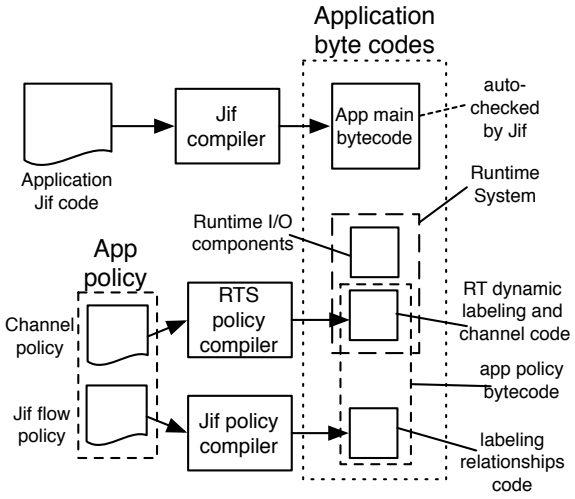

Figure 1. Infrastructure for compiling security policy into Jif applications for provable enforcement of policy.

depends on the structure of the data itself (the To: address on an email or the source address of a packet, e.g.) or 3) on a series of data exchanges (an authentication protocol, e.g.). 4) The protection offered on an output channel may depend on data (like an authentication token) that has been received on a companion input channel. 5) As the protection on an output channel may change over time, the output channel API must include a means to determine its current label.

These challenges guided the design for the `Channel`. Simultaneously, we ensured that the `Channel` would help the programmer meet our principles for sound RTS development. Firstly, a `Channel` maintains state between uses. By keeping track of some previous inputs and outputs, the `Channel` can track authentications and modify labeling policies based on transmitted data. This is critical for handling data-specific labeling (2) and authentication protocols (3). Also, when the system must be queried after a stream is opened to determine the security context (1), this context can be saved as part of the state of the `Channel`. Secondly, input and output streams are coupled together in a single `Channel` to allow input data to affect the labeling of outputs (4). In our experiments, we did not find a need for multiple inputs to be coupled with a single output or vice versa, although some channels could naturally have an input (or output) such as a read-only (write-only) file with no corresponding output (or input).

Another innovative quality of `Channels` is that they can be queried to determine what protection they offer for the next output. A `Channel` connected to a file may return a label indicating that it can protect data according to the security properties on the file.

Channels can operate at the level of semantically expressive Objects rather than only streams of bytes. They provide a specific object type as input and expect a specific object type (a String, a Packet, or some other data structure) as output. Hence, Channels can adjust I/O according to the proper semantic granularity demanded by the application. This is an important feature for handling the challenges of data-specific labeling (2).

Finally, a key design goal of the Channel abstraction was to make sure that Channels could be activated and configured by high-level policy. We have included a level of indirection for activating each kind of Channel, by not allowing Channels to be instantiated except through Policy. Another level of indirection allows customization of the labeling policy for each Channel input and output. Finally, external classes implementing authentication protocols can be activated to customize Channel instances in different settings.

3.2 Example

The policy given in Listing 6 defines the semantics for the standard I/O channel given in Section 3.1. It starts with public labels and allows the user to authenticate herself as having "stats" privileges. The StdioChannel contains the standard input and standard output streams. It prints Strings to standard out and retrieves Strings from standard in. Firstly, the channel must be enabled by adding a policy entry to the high-level policy file. The channel line of this policy (Listing 6, line 1) indicates that the StdioChannel should be enabled in the Policy and can be selected in the application with the String "stdio" (Listing 7, line 13). Without such a policy line enabling StdioChannel, it could not be used in the application.

Listing 6. Policy entries for Std. I/O channel.

```
1  channel policy "stdio" {
2    channel pol.StdioChannel
3    authentication "stats"
4    labeling "flowwall" }
5  authentication policy "stats" {
6    pol.StdioAuth.pwdauth ["stats-pwds.txt"] }
7  labeling policy "flowwall" {
8    init pol.StdioChannel.setPublicLabel
9    inputs pol.StdioChannel.getCurrentLabel
10   outputs pol.StdioChannel.getCurrentLabel }
```

The initial label on inputs and outputs is established by the method pol.StdioChannel.setPublicLabel as defined in the stdio's labeling policy's init field. This method establishes standard I/O as being public initially (this will correspond to the desired policy for the FLOWWALL in which no one is allowed to see statistics on packets until authenticated as having that privilege). The labeling policy for this channel is quite simple—the method getCurrentLabel merely returns the current label (which is kept as part of the state for the channel). To raise the security of the channel, an authentication module may be used in the application code, in this case causing pol.StdioAuthenticate.pwdauth to be called with a parameter, "stats-pwds.txt" indicating the location of the password file.

A StdioChannel may be used as previously shown in Listing 4 or as shown in Listing 8.

Listing 7. Policy class generated automatically from the policy in Listing 6.

```
1  public class Policy {
2    static public void authenticate(Channel channel,
3                                    String authType, Label l)
4    throws PolicyException {
5      if (channel instanceof StdioChannel &&
6                  authType.equals("stats"))
7        StdioAuth.pwdauth((StdioChannel)channel, l,
8                          "stats-pwds.txt");
9    }
10   static public Channel getChannel(String channelType,
11                                    Object params, Label l)
12   throws PolicyException {
13     if (channelType.equals("stdio"))
14       return StdioChannel.getInstance();
15     else return null;
16 }}
```

Listing 8. A simple use of standard I/O.

```
final Channel[{}] stdio =
            Policy.getChannel("stdio",null,new label{});
if (stdio != null) {
  final LabeledObject[{}] obj = stdio.getNextObject();
  if (obj != null) {
    String str = (String)obj.getObject();
    stdio.put("You entered " + str, obj.lbl);
}}
```

3.3 A principled runtime system

A design goal for the Channel and Policy classes was to guide programmers in implementing the principles given in Section 2.2.

Principle 1 Isolate dynamic labeling This is achieved by pushing all dynamic labeling decisions into the Channel classes. Each object is labeled as it enters the application, as determined by the semantics of the Channel class. Likewise, the Channel class limits the objects that can be output from an application, based on the object's label.

Principle 2 Limit the runtime API Because the runtime API is governed by what Channels can be retrieved from the Policy class, the Channels used by an application can be easily limited to what is needed.

Principle 3 Customize semantic granularity An alternative design for Channels is to restrict Channels only to read and write individual bytes. This fails to accomodate the needs each application has for a specific semantic granularity of inputs and outputs. On the contrary, our design

Listing 9. IPTables-style firewall rules

```
iptables −A FORWARD −d 192.168.1.20 −j DROP
iptables −A FORWARD −s 192.168.1.0/24 −d 192.168.2.0/24 −j ACCEPT
iptables −A FORWARD −s 192.168.2.0/24 −d 192.168.1.0/24 −j ACCEPT
```

requires `Channels` to get and put `Objects`, freeing the developer to design the semantic granularity as appropriate to the application.

Principle 4 Configure runtime labeling through high-level policy This principle is met through our policy infrastructure which allows high-level policy to be compiled into a `Policy` class that is specialized for each application. The `Policy` class governs which channels can be used in an application and how the channels can be authenticated. It also allows configuration of these decisions, specifying credentials repositories such as a keystore or password file.

In the following section we present FLOWWALL. We focus on extending the `Channel` class and policy infrastructure to handle the unique demands of this application.

4 FlowWall

To evaluate our design principles, we apply our approach to a real-world application, a network firewall. Network firewalls are a well-known part of the security infrastructure of almost every computer. Ensuring that a network firewall properly implements its policy is not always an easy problem [9], however, and benefits from automated assistance. The task of a network firewall is, essentially, to prevent illicit network packet flows across a particular boundary (a particular computer, an enterprise router, etc.).

A basic firewall policy signifies this with rules as in Listing 9 (we borrow a subset of the rule syntax used by the standard UNIX firewall tool, IPTables). Rules may contain a `−s` flag to indicate the source address (including optional port number[1]), a `−d` flag to indicate the destination address (including optional port number) and a `−j` flag to indicate whether the packets should be accepted or dropped. The rules are evaluated top-down. In the Listing, if a packet matches the first rule (i.e. its destination is IP address 192.168.1.20, any port), it is dropped and further processing stops. If it does not match this rule, it is processed by the last two rules. The last two rules match on a 24-bit subnet for both source and destination. The ranges that this notation represents are 192.168.1.0 – 192.168.1.255 and 192.168.2.0 – 192.168.2.255. In the next section we describe how to implement this application in an STL.

4.1 An information flow policy for a firewall

In contrast to normal software development processes, the first step in developing any security-typed application must be to determine what kind of information flow policy it will enforce. This design phase is critical in security-typed languages and constitutes one of the greatest challenges to security-typed application development. Haphazard setup results in extremely difficult programming, because it leads to many labeling conflicts, which usually induces a cascade of relabeling throughout the application. For the FLOWWALL we want to maintain a simple information flow policy: *Packets arriving from a given source address may only flow to their destination address if allowed by the firewall policy rules. All other packets are dropped.*

[1]Port numbers are left out here for simplicity of presentation, but we handle them in the natural way in FLOWWALL.

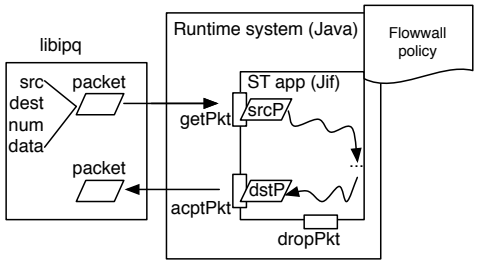

Figure 2. The FLOWWALL filters packets based on a high-level information flow policy.

As described in Section 2, application security analysis depends on (1) how system objects are dynamically labeled, (2) how labels are propagated on code throughout the application and (3) the relationships between labels (label semantics). We illustrate the basic structure of the FLOWWALL in Figure 2 such that these three areas can be identified. To determine whether this policy is fulfilled by FLOWWALL, it is necessary to determine (1) that input packets are labeled with their source addresses, (2) that the application type-checks and (3) that the relationships between source addresses and destination addresses are in compliance with the firewall policy rules. The STL compiler will automatically ensure that all code inside the inner box in Figure 2 is secure. This requires the security analyst only to check by hand the three API methods for the `PacketChannel`, in contrast to a normal firewall application, which would need to be checked entirely by hand.

As other functionality is added to examine or process packets, the Jif compiler will ensure that they maintain the security properties for packets as we showed earlier for `Stats.checkPacket` (Section 2.1). In this regard, we can treat additional modules as black boxes since they are guaranteed to sustain (or possibly somewhat restrict) the security properties established on data at their call sites.

We now present the design of FLOWWALL, focusing on the three elements which impact its overall security, the runtime system, application code and the high-level policy.

Listing 10. Main loop for getting, processing and accepting or dropping packets.

```
final Channel[{}] pktChannel = Policy.getChannel("packet",null);
...
while (true) { // code to handle one packet
  if (pktChannel != null) {
    final LabeledObject[{}] obj = pktChannel.getNextObject();
    // any processing of packets ...
    final label{} destL = pktChannel.getNextOutputLabel();
    if (obj != null) {
      Object pkt = obj.getObject();
      if (obj.lbl <= destL) // read "<=" here as "may flow to"
        pktChannel.put(pkt,destL);
}}}
```

4.2 Runtime system: Labeling inputs and outputs

The FLOWWALL needs a way to retrieve packets and a way to output packets. Packets that are not output are dropped. The first challenge is to implement the input labeling and output restrictions on packets using the `Channel` abstraction.

The key insight for implementing FLOWWALL is to use the source and destination addresses on packets as the security labels that will govern how packets can flow through the FLOWWALL. This insight is not imposed on the application but drawn out from the fact that the data flow semantics are really determined by the source/address pair in the packets. In light of this insight, the `PacketChannel`'s input channel delivers packets labeled with their source address. When a packet is retrieved from the `PacketChannel`, the channel's internal state reflects this by changing the label for the `PacketChannel`'s output channel—it will only accept a packet labeled with the proper destination address. This corresponds to the expected firewall policy that a packet can only be accepted if it can flow from its source address to its destination address. Having set up the runtime system I/O labeling, the final step is to configure the label semantics. The label semantics must reflect the firewall policy. If a source address `s` may flow to a destination address `d` in the firewall policy then the Jif label corresponding to `s` must be dominated by the Jif label corresponding to `d` in the label semantics (Jif flow policy).

This facilitates the code in Listing 10, which demonstrates the central processing loop of the FLOWWALL.

As expected, the API for packet channels meets the principles set forth in Section 2.2, because it utilizes the `Channel` pattern we established. Namely, Principle 1 is fulfilled by isolating all dynamic labeling of system objects (only the packets in this case) in the runtime infrastructure. Principle 2 is fulfilled by limiting the runtime API to include only the needed interface—for packets in this case. Principle 3 is fulfilled by basing the dynamic labels on source and destination addresses for the packets, which is the granularity of control desired for the application (not on data in the packets or any other criteria). Principle 4 is fulfilled by describing all possible address ranges for packets in the FLOWWALL policy and giving the licit flows by relating source and destination addresses. We describe these last two points, the granularity and configurability now in more detail.

4.3 High-level policy infrastructure

As shown in Figure 1, the two parts of high-level policy include an RTS policy governing the dynamic labeling of I/O and a Jif flow policy, defining the label semantics (and thus, the legal flows). The RTS policy for FLOWWALL must at least activate the `PacketChannel` with the labeling described above. The Jif flow policy must be a faithful encoding of the IPTables firewall policy. Then we implement the policy through the JP compiler [6] and our RTS compiler. Both policies compile into bytecodes that are linked into the application when it is executed. We now consider the advantages of this approach with regards to our principles, especially focusing on the last two principles: semantic granularity and high-level policy configuration.

4.3.1 Semantic granularity

Principle 3 asserts that the dynamic labeling provided in the runtime system should be targeted to a *semantic granularity* that is appropriate to the application. This refers to the amount of security context which must be determined before a label can be applied. One level of semantic granularity would be to consider all data input from the network merely as a stream of bits and label each bit the same way. This might be useful for enforcing a simple information flow policy that prevented network data from being saved to disk or printed to the screen, for example. Labeling network streams, disk streams and stdout with three different labels, {`network:`}, {`stdout:`} and {`disk:`} for example, and then ensuring that {`network:`} $\not\leq$ {`stdout:`} and {`network:`} $\not\leq$ {`disk:`} would be sufficient to achieve this information flow goal. In this case, there is no need to recover the semantic granularity of packets, source addresses and destination addresses.

As another example, we might be interested in a different level of semantic granularity which abstracts packets into SSL channels. In an instant messaging chat client, for example, we may want to label streams of String data with a label indicating the public-key certificate and certificate authority used when establishing the SSL connection.

Because the semantic granularity of labels depends on the security goals of the application, this cannot be achieved in a general way. Rather, the dynamic labeling runtime system for applications must be specialized based on the security goals of the applications and what level of information flows they seek to control. This motivates the important design goal of our policy and infrastructure: to simplify the

specialization process and improve reusability of RTS components.

For the FLOWWALL we are concerned with labeling packets as they enter the application and ensuring they have proper labels as they leave the application. Thus, labeling packets based on the context of their source and destination addresses is the best choice for the semantic granularity of labels. Namely, the security context of incoming packets consists of the source address and the security context of outgoing packets consists of the destination address. Whether a packet can flow from source to destination is governed by the firewall policy and enforced by the relationship between source labels and destination labels (label relationships are discussed in more detail below).

4.3.2 Establishing relationships between labels

Establishing the relationships between labels is critical for enforcing security policies over applications. The labels determine the expressiveness of the security policies that can be enforced. The relationships between the labels govern the ways that information can flow through an application. For example, in a traditional military security lattice [2], it is acceptable for unclassified information to flow into secret documents, but not vice versa. In this context, the label {`unclassified:`} is dominated by the label {`secret:`} allowing information flows from unclassified to secret. The establishment of relationships between labels should be done as much as possible in policy separate from the code, as stated in Principle 4.

The IPTables firewall policy must be isomorphic to the Jif policy used by the application. To ensure this is the case, we provide a specialized firewall policy compiler, just for use in FLOWWALL, that automatically generates the Jif policy from the IPTables policy. The basic conversion is that a source address label should be dominated by a destination address label exactly when the firewall rules allow packets from the source address to flow to the destination address. There are some subtleties in carrying this out, but since our main contribution is in the area of the RTS, not the labeling relationships, we reserve more details on the IPTables policy compiler to a companion technical report [7].

5 Evaluation

5.1 Analyzing FLOWWALL's security

A security analysis of the application is driven by the high-level policies for the RTS and the label semantics. The security goals for FLOWWALL were that it would accept or drop packets exactly in accordance with its IPTables firewall policy. Clearly, this depends on the application inputs, outputs and intermediate flows. The advantage of our approach is that all application inputs and outputs can be determined from the RTS policy. This reveals that packets enter and leave through the `PacketChannel`.

In order to determine how packets can flow through the application, we must know 1) how they are initially labeled, 2) how data with that label can flow through the application and 3) what labels must be on outputs. For determining input and output labels, we consult the RTS policy, which directs us to some code. This code must be carefully checked to ensure that packets are labeled with their source address. The RTS policy also directs us to code that reveals that the acceptable output labels for `PacketChannel` are always derived from the destination address of the last input. Finally, checking the Jif flow policy compiled automatically from the IPTables policy, we complete our analysis: 1) input packets are always labeled with their source addresses. 2) A packet labeled with its source address can only be relabeled to destination addresses allowed by the Jif policy and the Jif policy allows relabelings isomorphic to the IPTables firewall policy it was compiled from. 3) Only packets labeled with the destination address of the last input packet can be output; the rest are dropped.

In short, presuming the correctness of our compilers, we have a packet-filtering firewall which *provably enforces its policy*. This can be determined almost entirely without examining the application code, through inspection of the high-level policy and manual inspection of some labeling runtime infrastructure indicated by the policy. Inspection of the application code is only needed to ensure basic correctness: that the application actually reads any packets at all and does not just drop all packets.

This demonstrates our goal to show that security-typed languages significantly aid programmers in bridging the gap between high-level policy specification and code-level enforcement of that policy. At the same time, we have shown how runtime infrastructure plays a critical role in that implementation. Furthermore, using principled design, exemplified by our `Channel` pattern, we can allow the bulk of policy decisions to be deferred to an external, high-level policy.

5.2 Performance

Because performance is not our central consideration, we refer the reader to a more detailed performance evaluation in a companion technical report [7]. Our tests were run on an Intel 2.4 GHz Core 2 Duo with 2 GB of RAM, running Ubuntu Linux. In summary, FLOWWALL demonstrated performance expected of a Java-based application without any effort spent on optimization. At the same time, it demonstrated processing throughputs sufficient for an average Ethernet setting using gigabit Ethernet cards.

6 Related Work

There has been a great deal of prior research in STLs as described in the survey by Sabelfeld and Myers [11]. Much of this work has focused on laying theoretical foundations for STLs. Only two projects have generated realistic programming languages, Flow Caml [12] and Jif [10], and only

Jif has been used to build realistic applications. The number of realistic applications is still very small, but each would benefit from a principled, configurable runtime system as given in this paper.

JPmail [5], a PKI-based mail client, utilized a variety of system resources, but handled their security concerns in a limited way. Sockets were considered public and all dynamic labeling of emails was buried in the application code. Communications with password stores and keystores as well as user communication were accomplished in secure, but unprincipled ways that made it difficult to program and to verify the security of the application.

SIF [4] is a specialized runtime infrastructure for hosting web servlets. It provides a very limited API that controls how web servlets can interact with the web server and clients. Their infrastructure follows the first three of our principles, but provides no way to control or configure their I/O through high-level policy (principle 4). SIF is a reusable runtime system for a specific application area, but they give no insight about how to apply the concepts it utilizes to other application areas.

The runtime system for Security-enhanced Linux [8] is reusable and, because it interfaces with a MAC operating system, much security information is readily available for all system resources (all resources are already labeled with detailed security policies configured in a central policy). The interfaces developed for this project were ad hoc, however, and not configurable through high-level policy nor specialized for diverse semantic granularities.

7 Conclusion

The importance of security-typed languages for developing strongly secure systems has gained much credibility in the research community over the past few years. The importance of having strongly secure components whose security can be automatically checked and easily reasoned about promises to aid greatly the overall security of complex systems. Various obstacles have prevented them from being utilized widely, however, including the need for runtime systems which must be specialized per application and operating system environment and the need for high-level policy infrastructure which supports separation of security policy specification and implementation.

In this paper, we have identified the need for runtime system and high-level policy infrastructure, presented some principles that can be used to guide the development of such infrastructure for future applications and have demonstrated the utility of our principles on a small application, a network firewall. We have found these principles to be effective for developing the FLOWWALL. We are convinced that they will provide a good guide for future security-typed application developers and aid in the construction of secure systems.

Acknowledgements We are grateful for editorial help from Mike Hicks and from members of the Penn State SIIS Lab.

References

[1] A. Askarov and A. Sabelfeld. Secure implementation of cryptographic protocols: A case study of mutual distrust. In *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS '05)*, LNCS, Milan, Italy, September 2005. Springer-Verlag.

[2] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, MITRE Corp. MTR-2997, Bedford, MA, 1975. Available as NTIS AD-A023 588.

[3] W. Cheswick, S. Bellovin, and A. Rubin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, 2002.

[4] S. Chong, K. Vikram, and A. C. Myers. Sif: Enforcing confidentiality and integrity in web applications. In *Proceedings of the 16th USENIX Security Symposium*, Boston, MA, August 2007. To appear.

[5] B. Hicks, K. Ahmadizadeh, and P. McDaniel. Understanding practical application development in security-typed languages. In *22st Annual Computer Security Applications Conference (ACSAC)*, Miami, Fl, December 2006.

[6] B. Hicks, D. King, P. McDaniel, and M. Hicks. Trusted declassification: High-level policy for a security-typed language. In *Proceedings of the 1st ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS '06)*, Ottawa, Canada, June 10 2006. ACM Press.

[7] B. Hicks, T. Misiak, and P. McDaniel. Channels: Runtime system infrastructure for security-typed languages. Technical Report NAS-TR-0078-2007, Networking and Security Research Center, Department of Computer Science, Pennsylvania State University, 2007.

[8] B. Hicks, S. Rueda, T. Jaeger, and P. McDaniel. From trusted to secure: Building and executing applications that enforce system security. In *Proceedings of the USENIX Annual Technical Conference*, Santa Clara, CA, USA, June 2007. To appear.

[9] S. Kamara, S. Fahmy, E. Schultz, F. Kerschbaum, and M. Frantzen. Analysis of vulnerabilities in Internet firewalls. *Computers & Security*, 22(3):214–232, 2003.

[10] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java + information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001–.

[11] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[12] V. Simonet. FlowCaml in a nutshell. In G. Hutton, editor, *Proceedings of the first APPSEM-II workshop*, pages 152–165, Mar. 2003.

[13] S. Tse. *Dynamic Security Policies*. PhD thesis, University of Pennsylvania, 2007. Can be found at `http://www.cis.upenn.edu/~stse/main.pdf`.